Heterogenous Computing System for Deep Learning

Mihaela Maliţa[1] George Vlăduţ Popescu[2]
Gheorghe M. Ştefan[3]

[1]Saint Anselm College, Manchester, NH, USA
e-mail: mmalita@anselm.edu
[2]Politehnica University of Bucharest, Romania
e-mail: georgevlad.popescu@yahoo.com

[3]Politehnica University of Bucharest, Romania
e-mail: gheorghe.stefan@upb.ro

Abstract

Various forms of Deep Neural Network (DNN) architectures are used as Deep Learning tools for neural inspired computational systems. The computational power, the bandwidth and the energy requested by the current developments of the domain are very high. The solutions offered by the current architectural environment are far from being efficient. We propose a hybrid computational system for running efficiently the training and inference DNN algorithms. The system is more energy efficient compared with the current solutions, and achieves a higher actual performance per peak performance ratio. The accelerator part of our heterogenous system is a programmable many-core system with a Map-Scan/Reduce architecture. The chapter describes and evaluates the proposed accelerator for the main computational intensive components of a DNN: the fully connected layer, the convolution layer, the pooling layer, and the softmax layer.

Keywords: Deep Neural Network, parallel computing, heterogenous computing, accelerators.

1 Introduction

The mono-core computation can no longer keep up with the increasing demand of computational power requested by Deep Learning applications, making a multi- and many-core approach inevitable. At the same time, two kind of computations are segregated from the homogenous corp of the general purpose computing: the *complex computation* and the *intense computation*. The complex computation is defined by a code with the size, S, expressed as a number of lines, in the same magnitude order

with its execution time, T, expressed as a number of clock cycles. The intense computation is characterized by $S \ll T$. To optimize the power consumption, the execution time, and the area of chips, a solution based on a general purpose mono-core must be substituted with a heterogenous system. Whenever possible, the normal approach is to have for the complex computation a host engine, while for the intense computation an accelerator. The host engine could be a mono-core or a multi-core (multi means few) computational engine; however, the accelerator must be a many-core (many means no matter how big n) computational engine.

Our proposal for the accelerator part of the heterogeneous system includes a general purpose Map-Scan-Reduce Accelerator (MSRA) based on previous research [17] [13] [2], implementations [16], and investigated applications [14] [12] [1].

The main fallacy regarding the parallel computational systems currently used in Deep Learning applications is: the use of a gathering of consecrated processing cores, or of a specialized many-core engine, or of a specialized systolic array of circuits could be the solution for accelerating the DNN computation. Even if the use of an Intel's Many Integrated Core (MIC), or of an Nvidia's Graphic Processing Unit (GPU), or of a Google's Tensor Processing Unit (TPU) circuit, is ready to hand, the outcome will be inefficient because of various architectural incongruities. The architectural suitability that we propose allows to reduce $2-3\times$ the energy, and to increase approximately $\sim 3\times$ the *actual performance / peak performance* ratio.

In the following sections we describe the main computational requirements for DNN, the state-of-the-art hardware involved in Machine Learning applications, our proposed accelerator, and the implementation and evaluation of the main layers of a DNN.

2 The computational components of a DNN involved in deep learning

The computational components of a DNN are presented in this section. Two correlated issues challenge the implementation of the applications involving DNN: (1) the data transfer between the computational engine and the main memory of the system (unfortunately, the ghost of the *von Neumann Bottleneck* is still haunting us), and (2) the specific computations associated with the different types of DNN. We begin by addressing the second issue. The first issue will be discussed when the specific library of functions are defined and used in subsequent subsections 4.5 and 5.1.

Deep learning, as a branch of Machine Learning, uses various types of DNN. The most notorious are: Multi-Layer Perceptron (MLP), Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), Long Short Term Memory (LSTM), Deep Belief Network (DBN). The main computational layers to be accelerated in all the previous types of DNN are: the fully connected neural network layer, convolution layer, pooling layer and softmax layer. Simple vector operations must be also considered in order to articulate properly the layers just listed in order to obtain the more complex layers such as for RNN or LSTM.

2.1 Fully connected layers

A fully connected layer of n neurons receives an m-component input vector and sends out another n-component vector. The input vector, $\mathbf{x} = [x_1\ x_2\ \ldots\ x_m]$, is multiplied with a $m \times n$ matrix of weights and the result is submitted to a non-linear activation function (ReLU, sigmoid, ...).

Formally, the transfer function of a neuron is:

$$o = f(\sum_{i=1}^{m} w_i x_i) = f(net) \tag{2.1}$$

where f, the activation function, has various forms. For example:

- the sigmoid function of form:

$$f(y) = \frac{2}{1 + exp(-\lambda y)} - 1 \tag{2.2}$$

  where the parameter λ determines the steepness of the continuous function f; for a big value of λ the function f becomes: $f(y) = sgn(y)$

- ReLU (rectified linear unit) of form:

$$f(y) = max(0, y) \tag{2.3}$$

The neuron works as a combinational circuit performing the scalar product of the input vector $\mathbf{x}$ with the weight vector $\mathbf{w} = [w_1\ w_2\ \ldots\ w_m]$ followed by the application of the activation function. The activation function f, when it supposes a complex computation, is simply implemented using a look-up table (LUT).

A fully connected feed-forward NN is now a collection of n m-input neurons. Each neuron receives the same input vector $\mathbf{x}$ and is characterized by its own weight vector $\mathbf{w}_i$. The entire NN provides the output vector:

$$\mathbf{o} = [o_1\ o_2\ \ldots\ o_n] \tag{2.4}$$

The activation function is the same for each neuron. Thus, each NN is characterized by the weight matrix:

$$\mathbf{W} = \begin{bmatrix} w_{11} & w_{12} & \ldots & w_{1n} \\ w_{21} & w_{22} & \ldots & w_{2n} \\ \vdots & \vdots & \ldots & \vdots \\ w_{m1} & w_{m2} & \ldots & w_{mn} \end{bmatrix} \tag{2.5}$$

with each neuron having its own column of weights. The computation consists of a matrix-vector multiplication followed by the application of the activation function on each component of the resulting vector.

2.2 Convolution layer

Rather than using neurons to look at the entire input at a time, a convolution layer "scans" the input, crossing over the entire input with a small, $k \times k$, receptive field.

Let us consider the two-dimension input plan represented in the following matrix:

$$I = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ x_{21} & x_{22} & \dots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{p1} & x_{p2} & \dots & x_{pp} \end{bmatrix} \tag{2.6}$$

where x_{ij} are scalars (to make the story short and simple we considered a square matrix). For example, I represents the 8-bit pixels of one of the RGB plans associated with a color image. The image will be scanned looking each time to a $k \times k$ *receptive field* of the following form:

$$R_{ij} = \begin{bmatrix} x_{ij} & x_{i(j+1)} & \dots & x_{i(j+k-1)} \\ x_{(i+1)j} & x_{(i+1)(j+1)} & \dots & x_{(i+1)(j+k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{(i+k-1)j} & x_{(i+k-1)(j+1)} & \dots & x_{(i+k-1)(j+k-1)} \end{bmatrix} \tag{2.7}$$

Starting from the top left corner of the input plane, the first receptive field is R_{11}. Additional receptive fields are considered with a stride s horizontally and vertically:

$$i = 1, (1+s), (1+2s), \dots, (1+((p-k)/s)s) = 1, (1+s), (1+2s), \dots, (1+p-k)$$

$$j = 1, (1+s), (1+2s), \dots, (1+p-k)$$

where the stride could take values $s = 1, \dots, k$ (the stride cannot be bigger than k because the entire image must be scanned). If needed, the matrix I will be padded with zeroes to have $(p-k)/s = integer$.

The neuron is the same during the scan of the entire input plan. This is called a *filter* and is defined as a matrix having the same size with the receptive field. For each input plane d filters are defined:

$$F^y = \begin{bmatrix} f^y_{11} & f^y_{12} & \dots & f^y_{1k} \\ f^y_{21} & f^y_{22} & \dots & f^y_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ f^y_{k1} & f^y_{k2} & \dots & f^y_{kk} \end{bmatrix} \tag{2.8}$$

for $y = 1, 2, \dots, d$. Each filter investigates the input plane "looking" for a specific feature, thus generating a *Feature map* (see Figure 1). The filter F^y applied to the receptive field R_{ij} provides c^y_{ij} where:

$$c^y_{ij} = \sum_{m=1}^{k} \sum_{l=1}^{k} f^y_{lm} \times x_{(i+l-1)(j+m-1)} \tag{2.9}$$

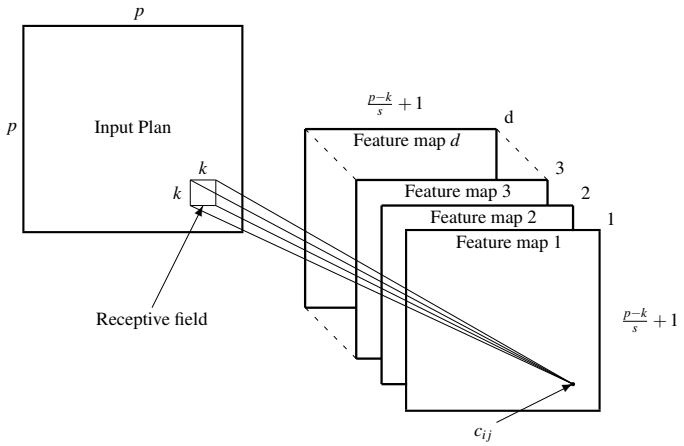

Figure 1: Convolution.

Thus, the application of the filter F^y with stride s provides the matrix:

$$C^y = \begin{bmatrix} c^y_{11} & c^y_{12} & \cdots & c^y_{1(((p-k)/s)+1)} \\ c^y_{21} & c^y_{22} & \cdots & c^y_{2(((p-k)/s)+1)} \\ \vdots & \vdots & \ddots & \vdots \\ c^y_{(((p-k)/s)+1)1} & c^y_{(((p-k)/s)+1)2} & \cdots & c^y_{(((p-k)/s)+1)(((p-k)/s)+1)} \end{bmatrix} \tag{2.10}$$

A convolutional layer consists of the application of d filters on the input plan generating a three dimensional array of $(((p-k)/s)+1) \times (((p-k)/s)+1) \times d$ scalars (see Figure 1). For each filter a feature plan is generated with a scalar for every receptive field.

2.3 Pooling layer

The pooling layer is used to reduce the size of a feature plan substituting (usually) a square *pooling window* of $q \times q$ scalars with only one scalar, which characterizes the entire pooling window. The scalar could be the maximum value from the pooling window, the sum of the values from the pooling window, or another value that is able to synthesize the content of the pooling window. The pooling windows are considered (usually) with a stride q in both directions in order to cover the entire feature plan. A stride smaller than q is possible, but it is not frequently considered.

Starting from a feature plan provided by a convolution, the pooling operation provides the pooled plan. Let us consider defining the pooling function with the same input I of $p \times p$ scalars. If the pooling window is $q \times q$ and the stride q (the usual case) the resulting plan is a $p/q \times p/q$ matrix of scalars P_q.

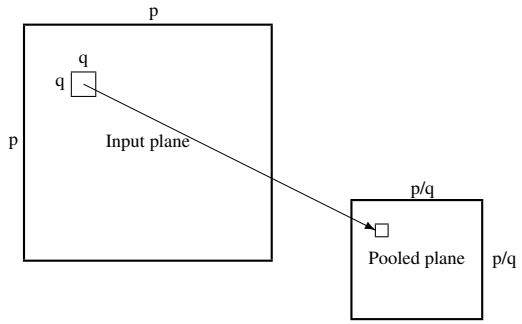

Figure 2: The pooling operation: starting from a $p \times p$ matrix, results in a $p/q \times p/q$ matrix.

$$P_q = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1(p/q)} \\ y_{21} & y_{22} & \cdots & y_{2(p/q)} \\ \vdots & \vdots & \ddots & \vdots \\ y_{(p/q)1} & y_{(p/q)2} & \cdots & y_{(p/q)(p/q)} \end{bmatrix} \tag{2.11}$$

where y_{ij} is computed usually in two ways (see Figure 3) for $q \times q$ matrices:

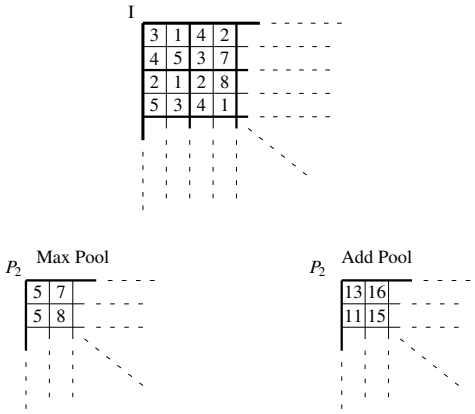

Figure 3: Examples of pooling for 2×2 pooling windows and stride 2. The Max Pool operation takes from the window the maximum value, while the Add Pool operation sums all the values from the window.

- by adding all the $q \times q$ values
- by taking the maximum value from the $q \times q$ values

Pooling window of $q \times q$ scalars in the matrix I results in a scalar in the P_q matrix (see Figure 2)

In current applications, the value of q is 2 or 4. In Figure 3 two examples for $q = 2$ are presented. One with *Max* as pooling function, and another with *Add* as pooling function. Each 2×2 matrix of scalars is substituted with their maximum or their sum.

2.4 Softmax layer

The softmax layer is used for multi-category classification, in order to emphasise the most probable candidate as result. It is applied to a n-component vector $V = \langle x_1, x_2, \ldots, x_n \rangle$. Its value is determined by the standard exponential function on each component, divided by the sum of the exponential function applied to each component, as a normalizing constant. Therefore, the output components sum to 1:

$$\sigma_i(V) = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} \tag{2.12}$$

Results:

$$S_i(V) = \langle \sigma_1(V), \sigma_2(V), \ldots, \sigma_n(V) \rangle \tag{2.13}$$

In [18] the computation is simplified by avoiding the divide operation and by reducing the domain of the exponent. The first step is to down-scale the exponentiation:

$$\sigma_i(V) = \frac{e^{x_i}/e^{x_{max}}}{(\sum_{i=1}^{n} e^{x_i})/e^{x_{max}}} = \frac{e^{x_i - x_{max}}}{\sum_{i=1}^{n} e^{x_i - x_{max}}} \tag{2.14}$$

The second step is to compute the natural logarithm:

$$ln(\sigma_i(V)) = (x_i - x_{max}) - ln(\sum_{i=1}^{n} e^{x_i - x_{max}}) \tag{2.15}$$

While the sum in Expression 2.12 is susceptible to overflow because the values generated by exponentiation are high, and the divide operation is resource and time consuming, the Expression 2.14 works with smaller numbers and avoids the division. Both, *ln* and *exp* operations are performed using LUTs.

2.5 Putting all together

An example of DNN is shown in Figure 4, where all the previously presented functions are used to define a particular network. The input is a color image represented by the three color plans RGB. A first convolutional level with ReLU as activation function is followed by a pooling layer which is used to downsize the feature plans. We follow similar stages (convolution with ReLU and pooling) until the last pooled volume is flattened to a vector applied to a fully connected NN. The last stage is a softmax layer which provides the probabilities associated with possible input images.

While the first few convolutions are used to inspect the input to identify specific *local* features, the last fully connected layers provide a *global* analysis, and the softmax output layer emphasizes the most probable result.

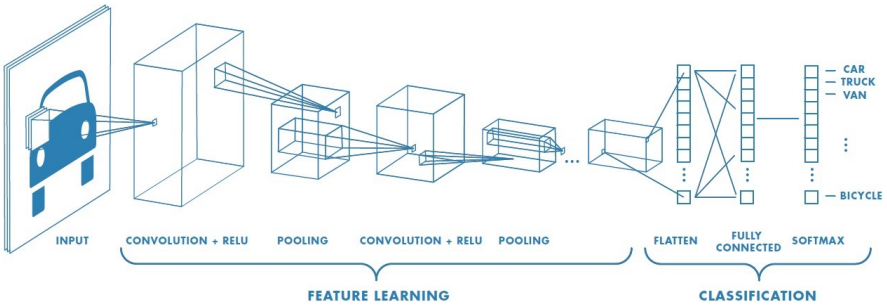

Figure 4: An example of DNN. [19]

3 The state of the art

The main drawbacks of the current hardware solutions are: (1) the programmable (CPU, MIC, GPU, DSP) solutions are implemented on inappropriate parallel engines unable to use efficiently their high computational power in the specific computation requested by DNN, (2) the specific circuit solutions do not have enough flexibility in order to be efficiently adapted to the various forms of DNN, and (3) the FPGA-based reconfigurable solutions are too expensive, consume too much energy and require hardware specific knowledge. In the following subsections we will review the main solutions based on "of-the-shelf" computational devices.

3.1 Intel's MIC

Central Processing Units were not initially designed for machine learning, but in the last few years manufacturers began including multiple processing units that allow parallel execution of different tasks, making them a good candidate for deep learning. **Intel Xeon Scalable**, is the first product based on Intel's MIC Architecture (see Figure 5).

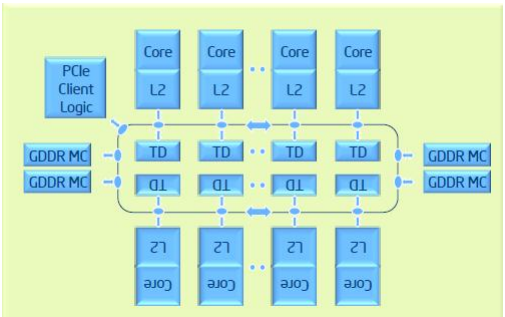

Figure 5: Xeon Phi Micro-architecture with 2 levels of cache and ring architecture [21]

Intel Xeon Platinum 8180 is a multi-core processor that offers a peak performance of 3259 GFLOPS for LINPACK [10] and 3.8 TFLOPS for SGEMM (Single precision floating General Matrix Multiply) using AVX512 [5]. The 28-core processor is fabricated using 14 nm technology and the TDP is 205 Watt, meaning a performance of 18.53 $GFLOPS/Watt$. This processor was evaluated in [5] for ResNet-50 topology and the obtained forward and backward propagation performance for the majority of the layers is 70%-80% of the machine peak. Nevertheless, there are layers where the performance is about 55% of the peak, because of the high bandwidth requirements for the process of writing the output tensors. An important observation is that the previous performance results were obtained for an optimized implementation called direct convolutional kernels. For other convolution implementation, such as `im2col`, `libxsmm` or `autovec`, the performance is much lower: 3 times smaller for `im2col`, 9 times smaller for `libxsmm` and 16 times smaller for `autovec`.

Intel Xeon Phi 7295 is a processor specialized for deep learning, with 72 cores and a peak performance of 11.5 TFLOPS for SGEMM using FMA4 instruction set. The processor is fabricated using 14 nm technology and the TDP is 320 Watt, meaning the FLOP/Watt performance is 35.9 $GFLOPS/W$.

This type of processor was also evaluated in [5] and the performance varied according to the filter dimensions. For example, layers with 1x1 filters achieve $\sim$ 50% of their peak performance and layers with 3x3 filters achieve 70% of their peak. For other convolutional layer implementations, the performance is even smaller than the one obtained for Xeon Platinum 8180: a $\sim$ 20% of peak average for `im2col` and just a few percent of peak for most of the layers, when using `libxsmm` or `autovec`.

Intel Xeon E5-2699 v3 (Haswell) CPU with 18 cores and a peak performance of 1.3 $TFLOPS$ for a TDP of 145 Watt (8.96 $GFLOPS/W$, 22nm technology) was evaluated in [8] for six DNN applications: two MLP networks, containing fully connected layers, two LSTM networks, containing fully connected and element-wise operation layers) and two CNN networks, containing convolutional, pooling and fully connected layers. The performance evaluation reveals that the CNN networks use 23% and 46% of processor peak computation capabilities, the MLP networks use 15.4% and 38.5% of processor peak computation capabilities and LSTM networks use 84.6% and 46% of processor peak computation capabilities.

3.2 Nvidia's GPU as GPGPU

GPUs are processors created for computer graphics (see Figure 6), which, due to their high number of cores and high parallelism, are very effective in running matrix multiplications, the main operation involved in deep learning.

The most famous GPU manufacturer is NVIDIA. Besides the usual GPUs, during the last years they have begun to make their products more efficient in artificial intelligence tasks.

Although the GPUs have large computational capabilities, the real performance obtained in deep learning applications may be far from their peak performance. The

Figure 6: NVIDIA Titan V [15]

factors that can influence the efficient use of computational and power resources are either related to processor architecture or software that is not optimal for the architecture it targets. This gap between theoretical and real performance for GPUs has been highlighted in several papers.

NVIDIA GTX Titan Black is a GPU with 2880 CUDA cores and 5645 GFLOPS single precision floating point (FP32) peak performance. The GPU was fabricated in 28 nm technology and the TDP is 250 Watt, which means a FLOP/Watt performance of 22.56 $GFLOPS/Watt$. The use of computational performance on CNN experiments are positioned in the range of 9% – 50% from peak [11].

NVIDIA Tesla K40 is a GPU with 2880 CUDA cores and 4.29 TFLOPS single precision peak performance (28 nm technology, TDP = 235 Watt, 18.25 $GFLOPS/W$). Different convolutional layers were tested and the obtained performance ranges from

23% to 35% of peak [3].

NVIDIA Geforce GTX 980 is a GPU with 2048 CUDA cores and a peak single precision performance of 4.95 TFLOPS (28 *nm* technology, TDP = 165 Watt, 30 $GFLOPS/Watt$). Different convolutional layers were tested and the obtained performance ranges from 30% to 51% for NVIDIA Geforce GTX 980 [3].

NVIDIA Tesla K80 is a card containing two GPUs, each one with 2496 CUDA cores and a peak single precision floating point performance of 2.8 TFLOPS (28 *nm* technology, TDP = 150 Watt, 18.66 $GFLOPS/Watt$). The evaluation for two MLP networks, two LSTM networks and two CNN networks reveals that the CNN networks use 32.1% and 35.7% of peak performance, the MLP networks use 7.14% and 25% of peak performance and LSTM networks use 17.85% and 25% of peak performance [8].

Although many applications require high precision computation (32-bit floating point FP32, or 64-bit floating point FP64), researchers have discovered that a half precision floating point (FP16) is sufficient for deep learning training. Additionally, deep learning inference can be performed using 8-bit integer computation, without significant impact on accuracy [6]. In order to make GPUs more efficient in performing different tasks, multiple precision modes are supported.

Starting with the Volta generation, a specialized Tensor Core unit was added, speeding up the matrix multiplications. Volta Tensor Cores combines FP16 (half floating point precision) multiplications with FP32 accumulations. The newer Turing Tensor Cores are enhanced for inferencing, adding new INT8 and INT4 precision modes. In order to evaluate the performance of GPUs using those specialized cores, a new performance metric was defined: Tensor Tera Operations Per Second, TTOPS.

Example of GPUs optimized for deep leaning are NVIDIA Titan V (see Figure 6) and NVIDIA Tesla V100. Titan V contains 5120 CUDA Cores and 640 Tensor Cores and offers a performance of 13.8 TFLOPS for single precision floating-point (FP32), 27.6 TFLOPS for half precision floating-point (FP16) and 110 TTOPS for deep learning. Tesla V100 (PCIe) contains 5120 CUDA Cores and 640 Tensor Cores and offers a performance of 14 TFLOPS for single precision floating-point (FP32), 28 TFLOPS for half precision floating-point (FP16) and 112 TTOPS for deep learning.

3.3 Google's TPUs

TPU is an Application-Specific Integrated Circuit (ASIC) for neural networks inference, used as accelerator in a hybrid system, the communication between TPU and host being assured by a PCIe bus.

The core of the chip (see Figure 7) is a systolic array of 256×256 8-bit multipliers, called Matrix Multiply Unit (MXU), which performs matrix multiplications between input data and weights. The MXU input data is stored in the Unified Buffer, which helds the results of previous computation steps. The data transfer between Unified Buffer and the host memory is controlled by a DMA controller. The MXU input weights are delivered by the Weights FIFO.

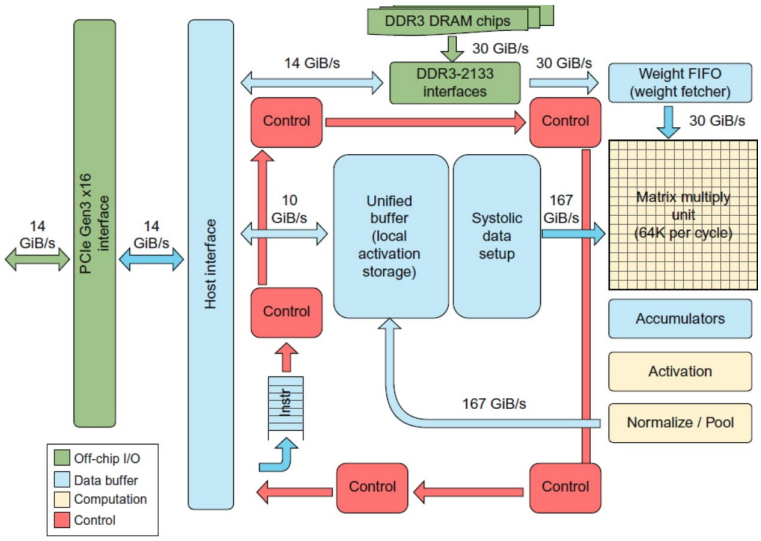

Figure 7: Block diagram of Tensor Processing Unit (TPU) [8]

The resulting products are accumulated in the Accumulators and the nonlinear activation functions are computed by the Activation Unit.

Running at a frequency of 700 MHz and computing 256×256 multiply-and-adds for 8-bit integers every clock cycles, the peak performance of TPU is 92 TTOPS. The chip was fabricated in 28 *nm* technology and the TDP is 75 Watt, meaning that the TTOPS/Watt performance is 1.22 $TTOPS/Watt$.

TPU was evaluated in [8] for the same six DNN applications described above. The MLP networks use 13.3% and 10.5% of its peak computing capabilities, LSTM networks use 4% and 3% of its peak computing capabilities and CNN networks use 93.4% and 15.3% of its peak computing capabilities, the mean performance being 23.24% of its peak. The performance for MLPs and LSTMs is limited by memory bandwidth, while the small performance of one of the CNNs networks can be explained by its structure (the presence of fully connected layers and the shallow feature depth of some layers).

Google also developed TPU v2 and TPU v3. If the initial TPU was limited to 8-bit integer operations, the new generations can also calculate in floating point (the MXU units perform multiplies at reduce `bfloat16` precision [20]), allowing it to be used also for neural networks training.

The second generation of TPU has two 128×128 MXUs, each one connected to an 8 GB High Bandwidth Memory, increasing the bandwidth to 600 GB/s. The peak performance for each TPU v2 chip is 45 TTOPS [9].

The third generation of TPU has two cores, each one with two 128×128 MXU units, peak performance being twice as the previous generation one.

3.4 Concluding about the state of the art

Putting together all the information from the previous analysis (see Table 1), we can learn some very important things about the way to define the structure and the architecture of an accelerator.

Table 1:

Chip model	**Fab. techn.**	**Peak performance**	**Performance per Watt**	**Actual % from peak**
Intel Xeon Platinum 8180	14 nm	3.8 TFLOPS	18.53 GFLOPS/W	5% - 80%
Intel Xeon E5-2699 v3	22 nm	1.3 TFLOPS	8.96 GFLOPS/W	15% - 84%
Intel Xeon Phi 7295	14 nm	11.5 TFLOPS	35.9 GFLOPS/W	20% - 70%
NVIDIA GTX Titan Black	28 nm	5.64 TFLOPS	22.56 GFLOPS/W	9% - 50%
NVIDIA Tesla K40	28 nm	4.29 TFLOPS	18.25 GFLOPS/W	23% - 35%
NVIDIA Geforce GTX 980	28 nm	4.95 TFLOPS	30 GFLOPS/W	30% - 51%
NVIDIA Tesla K80	28 nm	2.8 TFLOPS	18.66 GFLOPS/W	7% - 35%
Tensor Processing Unit	28 nm	92 TTOPS	1.22 TTOPS/W	3% - 93%

The general purpose architectures implemented by MICs provide a pretty good actual performance from the peak performance (an average of $\simeq 45\%$) but the computation per Watt is relatively low (an average of $\simeq 21\ GFLOP/Watt$). The reduced number of cores (less than 100) requests a simple control, allowing, in some applications for optimized code, the use of $> 80\%$ from the peak performance. But, most of the applications, evaluated for general purpose multi-cores, are unable to use more than 25% from their peak performance, and some of them use only few percentage of their very high performance. The architecture of the general purpose computers are designed for a wide specter of applications, while for intense applications there are specific requirements. The general purpose processors waste too much resources for 32-bit floating point computations, while usually the CNN computation asks small integer arithmetic for inferences and accepts 16-bit float operations for training.

General purpose graphic processing unit (GPGPU) is an oxymoron. It is tempting to take "of-the-shelf" a many-core parallel processor to solve intense computations, but, in the same time, is dangerous to use a powerful processor far from its application domain. A graphic machine can not be converted in a machine learning accelerator without a high risk. The actual performance related to the peak performance is lower compared with general purpose multi-core architectures (average $\simeq 34\%$) due to the difficulties involved in control and data transfer for hundred or thousands of execution units. The energy use is only a little improved (an average of $\simeq 22.36\ GFLOP/Watt$). The GPUs optimized for deep learning are designed with distinct physical resources for integer, 32-float, 64-float, and tensor operations (see Figure 6). Thus the area efficiency is lowered because too many times big parts of the area of the chip is unused. The cache approach persists with its inefficiency in helping the intense computation which is highly predictable requesting only a buffer-centered approach in the memory hierarchy design.

Specific ASIC's, such as TPU, do not have enough flexibility to support the high variety of DNNs. Therefore, only for a small part of the applications their huge computational power can be activated. For most of the applications (90%, according to [7]) no more than 13.4% from the peak performance is used. Only for one application, deployed in less than 5% of the applications, 93% from the peak performance is activated. The very big number of arithmetic systolic units (multipliers and adders) can not be easy put to work efficiently for the high variety of DNN we are facing in real applications.

The cache-based memory hierarchy is inappropriate for intense computation because of the high predictability of the program and data flow.

Architectural inadequacy is the main issue. A general purpose architecture or a graphic architecture or a simple systolic circuit are hard to be adapted to the specific requirements of the intense computational domain of machine learning. And when the energy saving criteria is added, the problem becomes much harder.

4 Map-Scan/Reduce Accelerator

4.1 The heterogenous system

The computation becomes "hybrid" or heterogenous when we start to segregate the execution of a program in two tightly interleaved parts:

- intense computations, characterized by short code and big execution time; these parts are sent to an accelerator
- complex computations, when the size of code and the execution time are in the same magnitude order; these parts run on the host.

Thus, the pair HOST & ACCELERATOR represents the structure of a HETEROGENEOUS COMPUTER. A possible embodiment is presented in Figure 8, where:

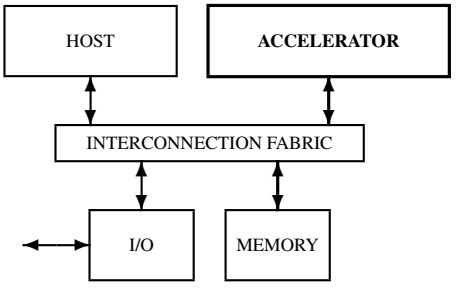

Figure 8: Heterogenous system.

- HOST is a general purpose processor which runs the application using a library of functions written using a kernel library running on ACCELERATOR (for example the Eigen library or the TensorFlow library implemented using EigenKernel or TensorFlowKernel libraries)
- ACCELERATOR is our MSRA running EigenKernel or TensorFlowKernel libraries, i.e., Eigen or TensorFlow libraries limited to data structures managed efficiently in a n-cell accelerator
- INTERCONNECTION FABRIC is a multiple point interconnection network used for fast data transfer between the components of the system
- MEMORY stores the programs and the data to be processed
- I/O system to connect the heterogeneous computer to the external world.

The host processor is programmed in a high level language (C, C++, Python, ...) while the accelerator's kernel library is developed, in this early stage of the project, in assembly language in order to achieve the highest possible performance. From the kernel library to the targeted library the implementation is done in a high level language.

4.2 The accelerator's structure

The structure of MSRA is presented in the current subsection. It is a general purpose programmable parallel accelerator optimized for the functional requirements described in Section 2. The accelerator is designed as part of a heterogenous computing system in which the complex part of the program runs on the host computer, while the intense part of the application (convolution, pooling, fully connected NN) runs on the accelerator. The architecture and the structure of the heterogenous system are described with emphasis on advantages provided for the investigated application domain.

MSRA is a n-cell engine (see MAP section in Figure 9) with two global loops:

- one, closed directly through a log-depth scan circuit, SCAN, which receives a n-component vector from the array of cells and sends back a n-component vector
- another, closed through a log-depth reduction circuit, REDUCE, which receives a n-component vector from the array of cells and sends its output to CONTROLLER which issues in each cycle, through the log-depth network DISTRIBUTE, an instruction to be executed in each active cell.

The architectural image of MSRA for the user is:

- the constant vector index, distributed along the cells: $IX = [1,\ldots,n]$ used to identify the cells as $cell_1, cell_2, \ldots, cell_n$
- the distributed memory:

$$DM = \begin{bmatrix} s_{11} & s_{12} & \ldots & s_{1n} \\ s_{21} & s_{22} & \ldots & s_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ s_{m1} & s_{m2} & \ldots & s_{mn} \end{bmatrix} \tag{4.1}$$

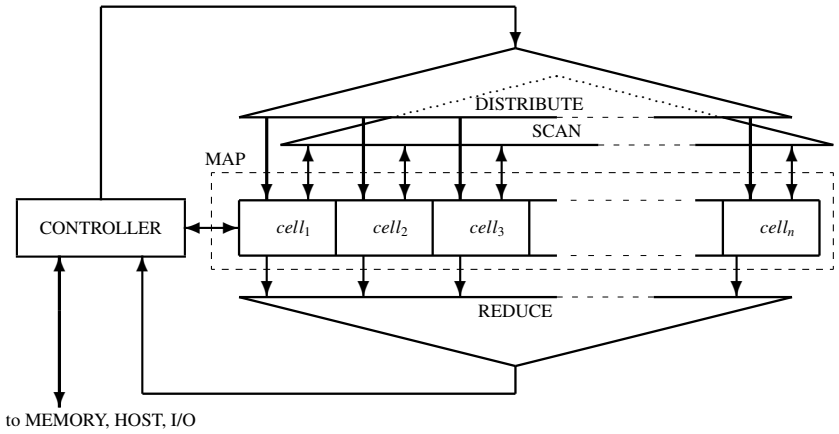

Figure 9: Map-Scan/Reduce Accelerator (MSRA): the linear array of cells MAP with two global loops (one through REDUCE and another through SCAN) running code issued in each clock cycle by the CONTROLLER unit.

which can be seen as composed by

- full *horizontal vectors*, distributed along the cells, for $i = 1, \ldots, m$:

$$H_i = \begin{bmatrix} s_{i1} & s_{i2} & \ldots & s_{in} \end{bmatrix} \tag{4.2}$$

  whose components can be processed in parallel in the MAP section of the accelerator
- full *vertical vectors*, for $j = 1, \ldots n$, each stored in the corresponding cell:

$$V_j = \begin{bmatrix} s_{1j} \\ s_{2j} \\ \vdots \\ s_{mj} \end{bmatrix} \tag{4.3}$$

  whose components are stored in the m-location register file in each cell

- the Boolean vector distributed along the cells: $B = [b_1, \ldots, b_n]$ used to activate the cells when $b_i = 1$
- the accumulator vector $ACC = [acc_1 acc_2 \ldots acc_n]$ distributed along the cells
- CONTROLLER's data memory: $M = [s_1 s_2 \ldots s_u]$
- the accumulator of CONTROLLER: *acc*

In the program memory of CONTROLLER, HOST loads programs whose binary form are stored as a pair of instructions, one for CONTROLLER and another to be executed in each active cell of the MAP array. Thus, in each clock cycle, from its program memory, CONTROLLER fetches an instruction for itself and another to be issued toward the MAP array. With a latency in $O(log\ n)$, CONTROLLER receives the result provided by the REDUCE network.

4.3 The micro-architecture

There are the following types of operations performed by MSRA:

- data transfer operations
- spatial control operations targeting the content of the vector B whose components are used to perform predicated executions
- unary and binary predicated vector operations on horizontal vectors
- reduction operations on the components of the horizontal vectors provided by the active cells
- scan operations on the components of the horizontal vectors provided by the active cells

4.3.1 Data transfer operations

The content of the distributed memory DM (Equation 4.1) can be partially or totally loaded from or stored to MEMORY (see Figure 8). The transfer is done one vector at a time using two functions:

- `load(s,i,j)`: s left most positions of the horizontal vector H_i are loaded with scalars from MEMORY starting at the address j
- `store(s,i,j)`: s left most scalars of the horizontal vector H_i are stored in MEMORY starting at the address j

Important note: the transfer between DM and MEMORY is transparent to the computation performed in our accelerator, i.e., during the transfer the computation runs in parallel undisturbed. This important feature of our architecture contributes to avoiding, at leas partially, the "bottleneck" between the MAP array and MEMORY. Smartly used, it feeds up the data hungry MAP array with data from MEMORY.

4.3.2 Spatial control operations

The spatial control allows the predicated execution by working on the value of the Boolean vector B. Each of the following operations is performed in one clock cycle:

- `activate`:
$$B \Leftarrow [1,1,...,1]$$
activates all the cells of the accelerator
- `where(cond)`:
$$b_i = ((b_i == 1)\ \&\ cond_i)\ ?\ 1\ :\ 0$$
in all active cells, keeps active only the cells where the condition $cond_i$ is fulfilled, where $cond_i$ stands for the value the condition `cond` takes in $cell_i$

- `elsewhere`: in all the cells active before the action of the last recent still lasting `where(...)`, keeps active only the cells where the condition $cond_i$ is not fulfilled, i.e., performs the complementary `where` selection in the active space
- `endwhere`: restores the vector B to the state before the last recent still lasting `where(...)`

Embedded `where(...)` are allowed.

Example 4.1 Let be, for $n = 16$ the boolean vector uninitiated and the accumulator vector loaded with the index vector IX:

```
B = [-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-]
ACC = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]
```

The evolution of the Boolean vector B under a sequence of spatial control operations is the following:

```
(1) activate;          => B = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
(2) where(acc > 3);    => B = [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1]
(3) where(acc < 14);   => B = [0,0,0,1,1,1,1,1,1,1,1,1,1,0,0,0]
(4) elsewhere;         => B = [0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1]
(5) endwhere;          => B = [0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1]
(6) endwhere;          => B = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

In step (1) all the cells are activated: $b_i = 1$, for $i = 1,2,\ldots,16$. Then, remain active only the cells where $acc_i > 3$. In the next step, the embedded `where` operation is exemplified: from the active cells remain active only the cells where $acc_i < 14$. The `elsewhere` operation is performed in the active space selected by the last recent `where(...)`: `WHERE(ACC > 3)`. Step (4) activates the cells inactivated by the previous `where(...)`. Step (5) restores the vector B to the state before the last recent `where(...)` where action still last: `WHERE(ACC < 14)`. In step (6) the action of `WHERE(ACC > 3)`, then vector B is restored to its initial value stated by the operation `ACTIVATE`.

◇

4.3.3 Arithmetic and logic operations

Each vector operation is performed on n-component horizontal vectors in constant time.

Unary Operations Some operations are performed only in the active cells where the value of the Boolean vector `B` is `bi=1`, other are performed in all cells, independent of the content of `B`. For example:

- `inc(i,k)`:

  $$s_{ij} \Leftarrow b_j \; ? \; (s_{kj} + 1) \; : \; s_{ij}$$

  for $j = 1,\ldots,n$; in each active cells, the component of the horizontal vector H_i takes the incremented component of the horizontal vector H_k

- `shiftRight(i,k,input)`:

$$s_{ij} \Leftarrow (j == 0)\ ?\ input\ :\ s_{k(j-1)}$$

for $j = 1,\ldots,n$; all the components of the horizontal vector H_k are shifted one position right with *input* inserted in the left end position, and are stored as H_i

Binary Operations The binary operations are performed in constant time (usually one clock cycle) only in the active cells where the value of the Boolean vector `B` is 1. For example:

- `mult(k,l,m)`:

$$s_{kj} \Leftarrow b_j\ ?\ (s_{lj} * s_{mj})\ :\ s_{kj}$$

for $j = 1,\ldots,n$; in each active cells, the component of the horizontal vector H_i takes the value of the product between the corresponding components of the horizontal vectors H_l and H_m

- `xor(k,l,m)`:

$$s_{kj} \Leftarrow b_j\ ?\ (s_{lj} \oplus s_{mj})\ :\ s_{kj}$$

for $j = 1,\ldots,n$; in each active cells, the component of the horizontal vector H_i takes the value of the bitwise XOR between the corresponding components of the horizontal vectors H_l and H_m

4.3.4 Reduction operations

The reduction operations are performed on the values provided by the active cells. They are:

- `redadd(i)`:

$$acc \Leftarrow \sum_{k=1}^{n} (b_k\ ?\ s_{ik} : 0)$$

the CONTROLLER's accumulator takes the sum of the accumulators of the active cells with a latency $\mathscr{L} \in O(log\ n)$

- `redmax(i)`:

$$acc \Leftarrow Max_{k=1}^{n} (b_k\ ?\ s_{ik} : 0)$$

the CONTROLLER's accumulator receives, with a latency $\mathscr{L} \in O(log\ n)$, the maximum value stored in the accumulators of the active cells

- `redmin(i)`:

$$acc \Leftarrow Min_{k=1}^{n} (b_k\ ?\ s_{ik} : \infty)$$

the CONTROLLER's accumulator receives, with a latency $\mathscr{L} \in O(log\ n)$, the minimum value stored in the accumulators of the active cells; the symbol ∞ stands for the biggest number in the representation used in the application.

- `redbool`:

$$acc \Leftarrow OR_{k=1}^{n} b_k$$

the CONTROLLER's accumulator receives, with a latency $\mathscr{L} \in O(log\ n)$, the logic OR from all the bits of the Boolean vector B (used to test if at least one cell is active or not).

4.3.5 Scan operations

The scan operations take a vector, HV_k, from LM and return a vector, HV_i, whose components are computed according to the global content of HV_k. For example:

- `prefixadd(i,k)`: $HV_i \Leftarrow [(b_1?s_{k1}:0), (\sum_{j=1}^{2}(b_j?s_{kj}:0)), \ldots, (\sum_{j=1}^{n}(b_j?s_{kj}:0))]$
- `compact(i,k)`: $HV_i \Leftarrow [s_{k1}, s_{k3}, \ldots, s_{k(p-3)}, s_{k(p-1)}, \underbrace{0,0,\ldots,0}_{n/2}]$ aligns to the left the odd components of the vector.

4.4 Hardware parameters of MSRA

The hardware performances are evaluated using simulation tools for the 28 nm technology. The simulation of a MSRA running at 1 GHz was used to evaluate the size and the power for a version having a 32-bit DDR interface, the 32-bit word size, the number of cells $n = 2048$, and the memory size $m = 1024$ (4KB SRAM/cell). The resulting area of the chip is $9.2 \times 9.2\ mm^2 = 84.64\ mm^2$. The power consumed by the chip is shown in Figure 10. Depending on temperature results: $12/14/18\ Watt$ at $80/100/120°C$. The computational performances are:

- for integer arithmetic:
  - 2048 GOPS, where GOPS stands for Giga 32-bit Operations Per Second; at $80°C$ results 170.66 $GOPS/Watt$
  - 4096 GOPS, for 16-bit operations; at $80°C$ results 341.33 $GOPS/Watt$
- for applications involving floating point arithmetic with 20% float operations plus 80% integer operations:
  - 820 GOPS for float and integer operations defined on 32 bits; at $80°C$ results 68.33 $GOPS/Watt$
  - 2400 GOPS for float and integer operations defined on 16 bits; at $80°C$ results 200.33 $GOPS/Watt$

  because the float operations are performed in a sequence of operations in execution units with no floating point units implemented as distinct structures, like in NVIDIA Titan V (see Figure 6)

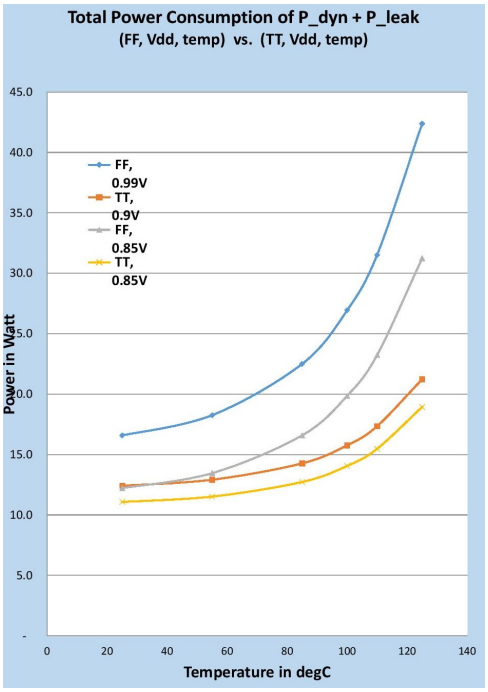

Figure 10: Power consumption evaluated for out MSA [14].

- for machine learning operations: 8 TTOPS, where TTOPS stands for Tensor TOPS (see Section 3.2); at 80°C results 1.36 $TTOPS/Watt$, if the architecture is designed for tensor operations used in Machine Learning applications[1].

To note that for training DNN the proposed architecture provides 200.33 $GOPS/Watt$, while for inference 341.33 $GOPS/Watt$ or 0.68 $TTOPS/Watt$ (around half of the circuit performance).

4.5 *NeuralKernel* library

For our application, a small library, let us call it *Neural*, can be implemented using the *NeuralKernel* library defined starting with the following functions:

`instmx(M,l,c)` : instantiate in ACCELERATOR's MAP array the matrix `M` with l lines and c columns, where the lines are parts of the full horizontal vectors (see Equation 4.2), and columns are parts of the full vertical vectors (Equation 4.3)

`instvt(V,l)` : instantiate in the data memory of ACCELERATOR's CONTROLLER the vector `V` of length l

`loadmx(M,p)` : load the content of the matrix `M` from MEMORY starting from the address `p` where the matrix is stored line by line, and increment the pointer `p` to `(p + l×c)`

[1]Tensor TOPS are evaluated (as for TPU) considering the REDUCE section performing adds in parallel with the MAP section which performs multiplications.

`storemx(M,p)` : store the content of the matrix `M` to MEMORY starting from the address `p` where the matrix is stored line by line, and increment the pointer `p` to `(p + l×c)`

`loadvt(V,p)` : load the content of the vector `V` from MEMORY starting from the address `p`, and increment the pointer `p` to `(p + l)`

`storevt(V,p)` : store the content of the vector `V` to MEMORY starting from the address `p`, and increment the pointer `p` to `(p + l)`

`mmm(M1,M2,M3)` : the matrix `M1`, is multiplied with the matrix `M2` and the result is stored as `M3`

`conv(M1,V,M2,k,s)` : the matrix `M1` is convoluted, with stride `s`, using a $k \times k$ filter stored line by line in the vector `V` and the resulting feature plan is stored as `M2`

`sigmoid(M1,M2)` : the activation function sigmoid is applied to the matrix `M1` with result as `M2`

`relu(M1,M2)` : the activation function ReLU is applied to the matrix `M1` with result as `M2`

`pooladd(M1,M2,q,s)` : the content of the matrix `M1` is pooled to the sum of the values of the $q \times q$ pooling window with stride `s`, and the result is stored as matrix `M2`

`poolmax(M1,M2,q,s)` : the content of the matrix `M1` is pooled to the maximum of the values of the $q \times q$ pooling window with stride `s`, and the result is stored as matrix `M2`

`softmax(M1,M2)` : the softmax function is applied to `M1` with result as `M2`

where the size of the matrices and vectors are limited by the parameters n, m, u previously defined.

5 Implementation and evaluation

Implementation and evaluation of the proposed system in implementing DNN are presented in this section. Roughly speaking, we use ***map*** resource to accelerate the convolution, the ***scan*** resource to accelerate the pooling, and the ***map-reduce*** resources for the fully connected NN. The algorithms we propose for the main computational patterns used in implementing a DNN are presented in versions easy to accelerate on our proposed architecture.

All the experiments we have done with the proposed architecture are performed on a cycle accurate simulator. The power estimate is based on the evaluation of the optimised synthesizable RTL. Therefore, in this stage of the project only the main functions of the *NeuralKernel* library are implemented and quantitatively evaluated.

5.1 Fully connected NN

The fully connected layer of a NN consists of a matrix-vector multiplication, the application of an activation function to the resulting vector, and the associated data transfer process. The algorithm is described in Figure 11, three processes run in parallel: (1) the control process, (2) the computation and (3) the data transfer.

```
/*****************************************************************
FUNCTION NAME: Fully Connected Neural Network
    - Load the weight matrix M starting with mxPointer
    - Multiply M with 'v' vectors stored successively starting with
        the address pointed by inVectPointer
    - Apply the activation function ReLU
    - Store the resulting 'v' vectors successively starting with the
        address pointed by outVectPointer
*****************************************************************/
instmx(M,line,column);      // define the size of the weight matrix M
instmx(V,1,column);         // define the size of the input vector V
instmx(W,1,column);         // define the size of the output vector W
inits(p1,mxPointer)         // initiate the value of matrix pointer
inits(p2,inVectPointer);    // initiate the input vector pointer
inits(p3,outVectPointer);   // initiate the output vector pointer

fullyConnectedNN(M,p1,p2,p3,n);
    loadmx(M,p1); // load in ACCELERATOR the content of M from p1
    loadmx(V,p2); // load in ACCELERATOR the first V starting from p2
    mmm(M,V,W);     // W <= M x V
    relu(W,W);      // W <= ReLU(W)
    load(V,p2);     // load the next V starting from p2 + c
    i <= 0;
    doInParallel{
        {   while (i < v-1)
            i = i+1;          } // control process runs on CONTROLLER
        {   mmm(M,V,W);
            relu(W,W);        } // computation running in MAP+REDUCE
        {   storemx(W,p3);
            loadmx(V,p2);     } // transferring process
    }
    storemx(W,p3);              // the last transfer of the result W
```

Figure 11: The algorithm for the fully connected neural network.

The execution time is dominated, for small v by the load of the weight matrix (`load(M,p1)`), or, for big v by the loop **doInParallel**. The application must be designed, if possible, so as too maximize the value of v. Thus, the matrix `M` is loaded only once for many uses.

For big v the execution time is dominated by the slowest process executed in parallel in the **doInParallel** loop. Because the transfer time is executed in time belonging to $O(c)$, we pay attention to the main computational process: {`mmm(M,V,W); relu(W,W);`}. In Figure 12, the algorithm for matrix-vector multiplication is presented. For `shiftRight(S,S,redadd(mult(V,M[i],V))` see Section 4.3 where the

multiplication, shift and reduction operations are defined. The execution time for this operation is $2+log_2 n$ when it is executed only once. For l executions, because of the pipelined hardware involved, the total execution time is $2 \times l + log_2 n$, where n is the number of cells in the MAP section of the ACCELERATOR. Thus, multiplying a $l \times c$ matrix with a c-component vector, with $c \leq n$, is executed in time belonging to $O(l)$, i.e., a n-cell ACCELERATOR provides an acceleration belonging to $O(n)$.

```
/*****************************************************************
FUNCTION NAME: Matrix-vector multiplication: W <= M * V
    - the matrix M with the lines M[i], for i = 1,2, ... l
*****************************************************************/
    for(i=1; i=<l; i=i+1)
        shiftRight(S,S,redadd(mult(V,M[i],V));
    for(i=1; i<(log_2 n); i=i+1)
        no operation;  // for the latency of the reduction add
    W <= S
```

Figure 12: Matrix-vector multiplication.

What is the constant associated to the acceleration in $O(n)$? The constant is for sure > 1, because in a mono-core processor the data transfer, the arithmetic operations and the control operation are performed sequentially, while in our architecture the **doIn-Parallel** loop is possible because specific hardware is provided for the three processes. We can claim that, for this function, the acceleration is supra-linear.

5.2 Convolutional layer

5.2.1 Stride $s = 1$

Step 1: load from the MEMORY the matrix I which is considered the input plan (Equation 2.6), as p lines (vectors) each of p components, and the filters as d sets of $k \times k$ scalars, then set $y \Leftarrow 1$, and $b \Leftarrow 1$. The execution time $t_1 = const \times (p^2 + d \times k^2)$.

Step 2: we consider the filter F^f as k "vertical" vectors:

$$F_i^f = \begin{bmatrix} f_{1i}^f \\ \vdots \\ f_{ki}^f \end{bmatrix}$$

for $i = 1, \ldots, k$, and the p "vertical" vectors:

$$I_j^b = \begin{bmatrix} x_{bj} \\ \vdots \\ x_{(b+k-1)j} \end{bmatrix}$$

for $j = 1, \ldots, p$, associated to the "band" b. Then we compute the matrix:

$$\begin{bmatrix} I_1^b \times F_1^1 & I_2^b \times F_1^1 & \ldots & I_p^b \times F_1^1 \\ I_1^b \times F_2^1 & I_2^b \times F_2^1 & \ldots & I_p^b \times F_2^1 \\ \vdots & \vdots & \ddots & \vdots \\ I_1^b \times F_k^1 & I_2^b \times F_k^1 & \ldots & I_p^b \times F_k^1 \end{bmatrix} = \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \end{bmatrix}$$

The computation is done in time $t_2 = 4k^2$ clock cycles.

Step 3: the following $(k-1)$ shift operations are applied to the last $(k-1)$ vectors:

$$\begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_k \end{bmatrix} => \begin{bmatrix} V_1 \\ V_2 << 1 \\ \vdots \\ V_k << (k-1) \end{bmatrix} = \begin{bmatrix} V_1' \\ V_2' \\ \vdots \\ V_k' \end{bmatrix} = \begin{bmatrix} v'_{11} & v'_{12} & \ldots & v'_{1p} \\ v'_{21} & v'_{22} & \ldots & v'_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ v'_{k1} & v'_{k2} & \ldots & v'_{kp} \end{bmatrix}$$

The operation is performed in $t_3 = 5(k-1)$ clock cycles.

Step 4: the line of the featured plan is computed as follows:

$$C_b = \begin{bmatrix} c_{b1} & c_{b2} & \ldots & c_{b(p-k+1)} \end{bmatrix}$$

where, for $i = 1, 2, \ldots, (p-k+1)$:

$$c_{bi} = \sum_{j=1}^{k} v'_{ji}$$

The computation is done $t_4 = k+1$ clock cycles.

Step 5:

if $b < p-k+1$ then $b \Leftarrow b+1$ and go to ***Step 2***.

Step 6:

The previous loop is repeated $p-k+1$ times providing the result of applying the filter F^y on the "image" I with stride $s = 1$:

$$\mathscr{C}^f = \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1(p-k+1)} \\ c_{21} & c_{22} & \ldots & c_{2(p-k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(p-k+1)(p-k+1)1} & c_{(p-k+1)(p-k+1)2} & \ldots & c_{(p-k+1)(p-k+1)} \end{bmatrix}$$

which is stored in the external memory. The execution of the transfer is in $t_6 = const \times (p-k+1)^2$.

Step 7:

if $f < d$ then $d \Leftarrow d+1$, $b \Leftarrow 1$ and go to ***Step 2***.

This loop is repeated d times resulting in the external memory a 3-dimension array of $(1+(p-k)/s) \times (1+(p-k)/s) \times d$ scalars.

The total execution time depends on the size of the input plan, the size of the filter and the number of features, and is:

$$t_{conv}(p,k,d) = t_{transfer} + t_{computation} =$$

$$t_{transfer} = (const \times (p^2 + dk^2 + d(p-k+1)^2)) \in O(dp^2)$$
$$t_{computation} = d(4(p-2)k^2 + (6p+10)k - 4(p+k^3+4)) \in O(pk^2d)$$

The convolution looks like an IO bounded function. In order to balance the data transfer with the computation we need to have a high bandwidth with MEMORY (the value of `const` must be small), and the overall DCNN computation must be organized with pooling layers applied before sending to MEMORY the feature plan. Thus, the weight of computation will be increased and the data to be transferred reduced.

5.2.2 Stride $s > 1$

For stride $s > 1$ the resulting 3-dimension array has the size:

$$(1 + (p-k)/s) \times (1 + (p-k)/s) \times d$$

because:

- vector C_i computed in ***Step 4*** is:

$$C_i(s) = [c_{b1} \underbrace{x \dots x}_{s-1} c_{b(s+1)} \underbrace{x \dots x}_{s-1} c_{b(2s+1)} \underbrace{x \dots x}_{s-1} \dots]$$

  where x stands for a meaningless value which must be removed from the final result

- in ***Step 5*** $b \Leftarrow b+s$ resulting in ***Step 6*** a $\mathscr{C}^f$ matrix with $1+(p-k)/s$ lines only:

$$\mathscr{C}^f(s) = \begin{bmatrix} c_{11} & x & \dots & x & c_{1(s+1)} & x & \dots \\ c_{(s+1)1} & x & \dots & x & c_{(s+1)(s+1)} & x & \dots \\ c_{(2s+1)1} & x & \dots & x & c_{(2s+1)(s+1)} & x & \dots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

The solution to provide a compact representation, by eliminating the xs, has two versions, one for the usual case when s is a power of 2 and another when s is of some value. In both cases we must add some sub-steps in ***Step 6***.

Version 1 for $s = 2^{integer}$: in ***Step 6*** we must use the scan operation `compact(i,j)` (see Section 4.3.5). If the stride is s then applying $log_2 s$ times the function `compact(i,j)` on each line of the matrix $\mathscr{C}^f(s)$ the meaningless values x will be eliminated.

The execution time for this step is:

$$t_{comp} = (1 + (p-k)/s) \times (2 + log_2 n) \times log_2 s$$

maintaining the execution time of ***Step 6*** dominated by the transfer.

Version 2 for s of some value : in ***Step 6*** we must add the following sub-steps:

Sub-step 6.1:

The matrix $\mathscr{C}^f(s)$ is transposed. Results a matrix $T(\mathscr{C}^f(s))$ with lines full of xs and lines containing only c_{ij} scalars.

Sub-step 6.2:

The matrix $T(\mathscr{C}^f(s))$ is compacted eliminating the lines of xs:

$$\mathscr{K}^f(s) = \begin{bmatrix} c_{11} & c_{(s+1)1} & c_{(2s+1)1} & \cdots \\ c_{1(s+1)} & c_{(s+1)(s+1)} & c_{(2s+1)(s+1)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Sub-step 6.3:

The matrix $\mathscr{K}^f(s)$ is transposed resulting the final result for the filter f

$$\mathscr{S}^f(s) = \begin{bmatrix} c_{11} & c_{1(s+1)} & c_{1(2s+1)} & \cdots \\ c_{(s+1)1} & c_{(s+1)(s+1)} & c_{(s+1)(2s+1)} & \cdots \\ c_{(2s+1)1} & c_{(2s+1)(s+1)} & c_{(2s+1)(2s+1)} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

which is a matrix s^2 times smaller than the matrix for $s = 1$.

The execution time for ***Step 6*** increases with

$$t_{6+} = t_{transpose} + 3p/s$$

Because, in our architecture $t_{transpose} \in O(p^2)$ the execution time of ***Step 6*** remains in the same magnitude order.

5.3 Pooling layer

The input for pooling is a matrix of type ***I*** (see Equation 2.6). The output is the matrix of

$$(1 + (p-k)/s) \times (1 + (p-k)/s)$$

components. The algorithm is similar to the one used for computing a feature plan in the convolutional layer for $s > 1$. Instead of a filter on the receptive fields are applied simpler functions. The differences are given by the lack of the filter, and occurs:

- in ***Step 2***, where instead of $I_i^b \times F_k^j$ is computed the sum $\sum_{i=b}^{b+k-1} x_{ji}$ or the maximum $MAX_{i=b}^{b+k-1} x_{ji}$
- in ***Step 4***, where is maintained $c_i = \sum_{j=1}^{k} v'_{ji}$ or is computed $c_i = MAX_{j=1}^{k} v'_{ji}$.

Thus, the computation time is evaluated in the same way, but refers to smaller input matrices.

5.4 Softmax layer

For the softmax layer the exponential and logarithmic functions are computed using LUTs because the accuracy offered by this way is enough in the domain of NN. In our implementation we use a LUT for the logarithm, `logLUT`, stored in the data memory of CONTROLLER, and another LUT for exponentiation, `expLUT`, replicated in each of the n cells of ACCELERATOR.

```
/*********************************************************************
ALGORITHM NAME: Softmax
    - input: V = <x1, ..., xp>
    - output: V = <sigma_1(V), ..., sigma_p(V)>
*********************************************************************/
(1)     V <= V - redmax(V);
(2)     V <= V - lnLUT(redsum(expLUT(V)));
(3)     V <= expLUT(V);
```

Figure 13: The softmax algorithm.

According to the solution presented in Section 2.4 the algorithm on our accelerator is shown in Figure 13. The execution time is $t_{softmax} = 9 + 4log_2 n$. The loop $MAP \rightarrow REDUCE \rightarrow CONTROLLER \rightarrow MAP$ is closed two times, and both, the reduction and the distribution network are log-depth. In step (1) of the algorithm CONTROLLER sends back to the MAP array, through a log-stage pipe, the maximum value of the vector `V` received from a log-depth circuit from the MAP array. In the second step of the algorithm, `redsum` is received by CONTROLLER in log-time, and the logarithm is sent back in the same time to the array. Thus the acceleration of this layer is in $O(n/log\ n)$.

The latency introduced by the reduction operations can be avoided, as we did for matrix-vector multiplication, if the function is applied to a stream of vectors, $[V_1, V_2, \ldots, V_u]$, accumulated in the local memories distributed along the cells of the array. Then the values for $max_i = redMax(V_i)$ can be stored in the data memory of CONTROLLER. If $n \sim u$ then this computation is done in $O(u)$ time avoiding the contribution of $log_2 n$ for each max_i. Similarly, the values for $redsum(expLUT(V_i))$ are treated. Thus, the computation will be accelerated by $O(n)$.

6 Conclusions

1. The acceleration provided for each layer is in $O(n)$ for a n-cell accelerator. Sometimes, the constant associated to $O(n)$ is > 1, i.e., the acceleration is supra-linear, because the control, the transfer and the computation are done in parallel (see subsection 5.1).

2. For MSRA, $\alpha = actualPerformance/peakPerformance$ in performing the computations associated to the stages of DNN is very high. Usually, $\alpha > 0.8$.

3. The data transfer between ACCELERATOR and MEMORY is transparent to the computational process. Thus, the effect of the bottleneck between ACCELERATOR and MEMORY is reduced.

4. The power consumption in our ***programmable system*** is 680 $TGOPS/Watt$[2] not far from the power consumption per Tensor operation provided by the TPU ***circuit***.

GFOPS/Watt is $2\times$ higher than for many-cores, and $3\times$ higher than for multi-cores.

4. But, we must pay attention to how the computational layers are interleaved with the data transfer stages in the implementation of an actual DNN. The main advantage in this respect for our architecture is the local memory in each cell of the MAP section. If this memory is big enough, then some data transfers can be avoided. Another advantage for our architecture is the possibility to transfer data in parallel with the computational process (see the algorithm for matrix-vector multiplication in Figure 12). The overall α coefficient, taking into account also the transfers between the local memory in cells and MEMORY (see Figure 8), decreases a little if the algorithms do ignore the possibility of transparent transfers.

5. Because MSRA has few characteristics similar to the Streaming SIMD Extensions (SSE) we must emphasize that the main differences consist of:

1. the control at the level of MSRA
2. the predicated execution according to the local state of each cell
3. the scan & reduction mechanism, which allows fast and efficient global vector to vector and vector to scalar operations
4. the large amount of local storage at the cell level, instead of the limited register file system in SSE
5. data and programs in MSRA are provided through simple buffer-like memories, due to the high predictability of their content, unlike in the SSE system where data and programs are provided through the area and energy consuming cache memory system
6. search and scan instructions in the MSRA approach help advanced stream, list or sparse matrix operations.

References

[1] Andonie, R., Malita, M.: The Connex Array as a Neural Network Accelerator. In: Proceedings of the IASTED International Conference on Computational Intelligence, Banff, Alberta, Canada, July 2–4, pp. 163–167 (2007)

[2]This peak performance is achievable taking in consideration the operations performed simultaneously in the MAP section and in the REDUCE section (see Figure 9). This happens, for example, when matrix-vector multiplication is performed.

[2] Andonie, R., Malita, M., Stefan, M. G.: MapReduce: From Elementary Circuits to Cloud, In: Kreinovich V. (ed.): Uncertainty Modeling, series: Studies in Computational Intelligence, Series Ed.: Kacprzyk J., Springer pp. 1–14 (2017)

[3] Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning. Available at: `https://arxiv.org/pdf/1410.0759.pdf` (2014)

[4] Cireşan, D. C., Meier, U., Masci, J., Gambardella, L. M., Schmidhuber, J.: Flexible, high performance convolutional neural networks for image classification. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence, pp. 1237–1242 (2011)

[5] Georganas, E., Avancha, S., Banerjee, K., Kalamkar, D.D., Henry, G., Pabst, H., Heinecke, A.: Anatomy of high-performance deep learning convolutions on SIMD architectures. SC. Available at: `https://arxiv.org/pdf/1808.05567.pdf` (2018)

[6] Harris,, M.: Mixed-Precision Programming with CUDA 8, Available at: `https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/` (2016)

[7] John L. Hennessy, J. L., Patterson, D.: Computer Architecture. A Quantitative Approach. Sixth Edition, Morgan Kaufmann, (2019).

[8] Jouppi, N. P., Young, C., Patil, N., Patterson, D., et al: In-Datacenter Performance Analysis of a Tensor Processing Unit^{TM}. In: 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 26. Available at: `https://drive.google.com/file/d/0Bx4hafXDDq2EMzRNcy1vSUxtcEk/view` (2017)

[9] Kennedy, P.: Case Study on the Google TPU and GDDR5 from Hot Chips 29, Available at: `https://www.servethehome.com/case-study-google-tpu-gddr5-hot-chips-29/` (2017)

[10] Kumar, A., Trivedi, M.: Intel Scalable Processor Architecture Deep Drive, Available at: `https://en.wikichip.org/w/images/0/0d/intel_xeon_scalable_processor_architecture_deep_dive.pdf` (2017)

[11] Li, C., Yang, Y., Feng, M., Chakradhar, S., H. Zhou, H.: Optimizing Memory Efficiency for Deep Convolutional Neural Networks on GPUs, SC '16. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, pp. 633-644 Available at:: `https://arxiv.org/ftp/arxiv/papers/1610/1610.03618.pdf` (2016)

[12] Lorentz, I., Malita, M., Andonie, R.: Evolutionary Computation on the Connex Architecture, In: Proceedings of The 22nd Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2011), pp. 146–153 (2011)

[13] Malita, M., Stefan, G. M., Thiébaut, D.: Not multi-, but many-core: Designing integral parallel architectures for embedded computations. In: ACM SIGARCH Computer Architecture News, 35(5) pp. 32–38 (2007)

[14] Malita, M., Stefan, G. M.: Map-Scan Node Accelerator for Big-Data. In: 2017 IEEE International Conference on Big Data (BIGDATA), 4th Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery, Dec. 11-14, 2017 Boston, MA, USA, pp. 3442-3447 Available at: `http://users.dcae.pub.ro/~gstefan/2ndLevel/technicalTexts/S18203.pdf` (2017)

[15] Smith, R., Oh, N.: The NVIDIA Titan V Preview – Titanomachy: War of the Titans. In: AnandTech. Available at: `https://www.anandtech.com/show/12170/nvidia-titan-v-preview-titanomachy/2` (Dec. 20,, 2017)

[16] Stefan, G. M., Sheel, A., Mîţu B., Thomson, T., Tomescu, D.: The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing, *Stanford University: Hot Chips: A Symposium on High Performance Chips*, August 2006. [Online]. Available: `https://youtu.be/HMLT4EpKBAw` at 35:00 (2006)

[17] Stefan, G. M., Malita, M.: Can one-chip parallel computing be liberated from ad hoc solutions? A computation model based approach and its implementation. In: 18th Inter. Conf. on Ciruits, Systems, Communications and Computers, pp. 582–597 (2015)

[18] Yuan, B.: Efficient hardware architecture of softmax layer in deep neural network. In: 2016 29th IEEE International System-on-Chip Conference (SOCC), pp. 323-326 (2016)

[19] Convolutional Neural Network. 3 things you need to know. Available at: `https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html` (2019)

[20] Cloud TPU. System Architecture. Available at: `https://cloud.google.com/tpu/docs/system-architecture` (2019)

[21] The Future is Hybrid Trends in Computing Hardware. Available at: `https://www.xcelerit.com/resources/the-future-is-hybrid-trends-in-computing-hardware/` (2017)